Mining Data Streams under Dynamicly Changing Resource Constraints

Conny Franke
Department of Computer Science,
University of California at Davis, USA

Marcel Karnstedt **Kai-Uwe Sattler**
Department of Computer Science and Automation,
TU Ilmenau, Germany

Abstract

Due to the inherent characteristics of data streams, appropriate mining techniques heavily rely on window-based processing and/or (approximating) data summaries. Because resources such as memory and CPU time for maintaining such summaries are usually limited, the quality of the mining results is affected in different ways. Based on *Frequent Itemset Mining* and an according *Change Detection* as selected mining techniques, we discuss in this paper extensions of stream mining algorithms allowing to determine the output quality for changes in the available resources (mainly memory space). Furthermore, we give directions how to estimate resource consumptions based on user-specified quality requirements.

1 Introduction

Stream mining has recently attracted attention by the database as well as the data mining community. The goal of stream mining is a fast and adaptive analysis of data streams, i.e., the discovery of patterns and rules in the data. An important task of traditional mining as well as stream mining is frequent itemset mining, that aims to identify combinations of items which occur frequent in a given sequence of transactions. Typical applications of mining data streams are click stream analysis, analysis of records in networking and telephone services and analysis of sensor data among others.

Another popular problem, especially for continuous data streams, is the detection of changes in the data. This includes aspects such as changes in the distribution of the data (possibly expressed in statistical terms like median or quantiles) and burst detection, and also task specific change detection, e.g., recognizing changes in the set of frequent itemsets, in the frequency of particular itemsets or changing correlations between itemsets. Concerning the problem of change detection, the challenge of in-time processing and signalizing gains additional importance beside resource restrictions.

The main challenge in applying mining techniques to data streams is that a stream is theoretically infinite and therefore in most cases cannot be materialized. That means that the data have to be processed in a single pass using little memory. Based on this restriction and the goals of data mining one can identify two divergent objectives: On the one hand the analysis should produce comprehensive and exact results and detect changes in the data as soon as possible. On the other hand the single pass demand and the problem of resource limitations allow only to perform the analysis on an approximation of the stream (e.g., samples or sketches) or a window (i.e., a finite subset of the stream).

However, using an approximation or a subset of the stream affects the quality of the analysis result: The mining model differs from the mining model we would get if the "whole" stream or a larger subset is considered. This is particularly important because some of the proposed stream mining approaches support time sensitiveness (reducing the influence of outdated stream elements) by using weaker approximations for outdated elements. Thus, the mining quality for these elements is worse than for newer data. The problem of the quality of the mining model can also be considered in the opposite direction: Based on user-specified quality requirements one could derive resource requirements, i.e., the memory needed for managing stream approximations in order to guarantee the requested quality.

We propose resource awareness in conjunction with quality awareness as one of the main requirements in stream mining – and challenges in parallel. Fig. 1 illustrates how the dependencies between both are integrated into the operational flow of stream processing. Two ways of putting resource and quality awareness into practice get evident:

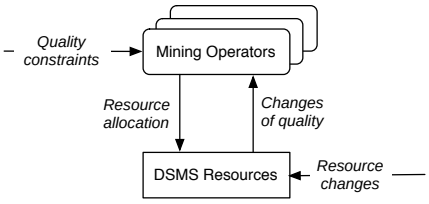


Figure 1: Mapping of quality

1. Claim for specific quality requirements and deduce the needed resources to achieve this quality.
2. Limit the resources provided for processing and deduce the achievable quality.

Based on this observation, we propose a resource-adaptive and quality-aware mining approach for frequent itemset mining. We argue that quality awareness is basically orthogonal to the specific mining problem, even if the individual mining approach requires dedicated techniques for considering mining quality. For this purpose, we discuss the general applicability of our approach and outline specific characteristics that potential mining techniques have to meet.

The remainder of this paper is structured as follows. After a brief survey of related work in Section 2 we introduce relevant quality measures in Section 3. In Section 4 we present our extended frequent itemset mining approach by adding quality measuring as well as a resource adaption based on user-specified quality requirements. Based on

this, we discuss approaches for resource-adaptive change detection in Section 4.3. After discussing the applicability of our approach to general mining problems in Section 4.4, we report results of an experimental evaluation in Section 5. Finally, we conclude the paper and discuss open issues for future work in Section 6.

2 Related Work

In this paper, we discuss strategies for adaptively mining data streams in general. We will develop our considerations on the basis of a corresponding approach for Frequent Itemset Mining proposed in [Franke *et al.*, 2005].

Frequent itemset mining deals with the problem of identifying sets of items occurring together in so-called transactions frequently. Any itemset occurring in more than σ (the *support*) percentage of all input transactions is regarded as frequent. Usually, a deviation of ε in the support of an itemset is accepted. σ and ε are (optionally dynamicly) user-defined parameters. Basically, two classes of algorithms can be distinguished: approaches with candidate generation (e.g., the famous apriori algorithm) as well as without candidate generation. Here, only the latter ones are suitable for stream mining. Usually, these approaches are based on a prefix-tree-like structure. In this tree – the frequent pattern (FP) tree – each path represents an itemset in which multiple transactions are merged into one path or at least share a common prefix if they share an identical frequent itemset [Han *et al.*, 2000]. For this purpose, items are sorted as part of their transaction in their frequency descending order and are inserted into the tree accordingly. Again, the FP tree is used as a compact data summary structure for the actual (offline) frequent pattern mining (the FP-growth algorithm).

In order to mine streaming data in a time-sensitive way an extension of this approach was proposed [Giannella *et al.*, 2003a]. Here, so-called tilted time window tables are added to the nodes representing window-based counts of the itemsets. The tilted windows allow to maintain summaries of frequency information of recent transactions in the data at a finer granularity than older transactions. The extended FP tree, called pattern tree, is updated in a batch-like manner: incoming transactions are accumulated until enough transactions of the stream have arrived. Then, the transactions of the batch are inserted into the tree. For mining the tree a modification of the FP-growth algorithm is used taking the tilted window table into account. The original approach assumes that there is enough memory available to deliver results in any required quality (expressed in terms of σ and ε, see Section 3) and no way is described how to proceed if the algorithm runs out of memory. In Section 4.2 we will discuss how the amount of required memory can be adjusted and how this affects the result quality.

In recent time, the idea of processing streams while adapting to resource and quality constraints gained boosted attention. There are several recent works dealing with quality-aware online query processing applications in centralized and distributed systems, e.g., [Berti-Équille, 2006]. But, only a couple discusses concrete approaches and algorithms in stream mining scenarios, least of all the conjunction of resource adaptiveness and quality awareness.

In [Gaber *et al.*, 2003] the authors propose a resource-adaptive mining approach based on similar goals as our work. They suggest the adaptation to memory requirements, time constraints and data stream rate by solely adapting the produced output granularity. The paper focuses on clustering, but the authors state the applicability to classification and frequent item mining (but not itemset mining). In succeeding works, e.g., [Gaber and Yu, 2006], they extend their approach by adding resource adaptiveness to the input rate and the actual data mining algorithm as well. However, the approach lacks providing quality guarantees for the mining results.

The algorithm RAM-DS (Resource-Aware Mining for Data Streams) proposed in [Teng *et al.*, 2004] uses a wavelet-based approach to control the resource requirements. It is mainly concerned with mining temporal patterns and the method can only be used in combination with a certain regression-based stream mining algorithm proposed by the same authors. Although the proposed algorithm for mining temporal patterns is resource-aware, it is not resource-adaptive and does not provide guarantees for the quality of the mining results.

Q	**Output**	**Factors**
Q_{Ma}	ε/σ	queried time interval
Q_{Mi}	σ	-
Q_{Tr}	maximal "look back time"	-
Q_{Tg}	minimal granularity	queried "look back time"
Q_{Tc}	minimal time till detection of changes	update interval u

Table 1: Quality measures and influencing factors

3 Quality Measures for Stream Mining

As stated in section 1, any resource adaptiveness comes along with effects on the achievable result quality. In [Franke *et al.*, 2005] we introduced different quality measures we examine in the context of stream mining. As a result, we distinguish several different classes of quality measures, which are summarized together with exemplarily chosen representatives in Table 1. For a more detailed discussion we refer to [Franke *et al.*, 2005].

All Q_{T*} are identical for different mining problems and symbolize concrete measures, while Q_{M*} represent classes of quality measures that are always specific to the investigated problem and the applied algorithm(s). A special measure for the problem of frequent itemset mining is the ratio between a value ε and the support σ, which reflects the maximal deviation from the defined support each of the itemsets finally identified as frequent could possess. In the context of frequent itemsets the support is a so called interesting measures Q_{Mi} ([Tan and Kumar, 2000]).

From our point of view, any mining techniques applied on continuous and evolving data streams should take time sensitiveness into account, thus, we define time as another important quality measure. Q_{Tr} describes how far we can look back into the history of the processed stream and Q_{Tg} how exact we can do this, which means which time granularity we can provide. Q_{Tc} corresponds to one of the main challenges of stream mining: the actual time we need in order to register changes in the stream. These temporal quality measures must not be confused with temporal aspects that influence the methodical quality (see Table 1).

For the remainder of this paper, if we refer to all quality measures as a whole, we will use the symbol of the superclass Q and the general term 'quality'.

This work aims for determining two (theoretical) kinds of functions:

1. $r : Q \rightarrow R$ - maps claimed quality to the resources needed, and
2. $\overline{r} : R \rightarrow Q$ - maps provided resources to the best achievable quality, as an inverse function to r.

More detailed, r is one function $r(args_x, Q_x(args_x), args_y, Q_y(args_y), \ldots)$ taking all claimed quality measures $Q_x, Q_y, \ldots$ and their factors as input, but we write $r(Q)$ for short. In contrast, $\overline{r}$ represents a bundle of inverse functions $\overline{r_x}, \overline{r_y}, \ldots$, each corresponding to one quality measure $Q_x, Q_y, \ldots$. Moreover, as we do not state the distribution of the stream elements as input factor for any function, r and $\overline{r}$ differ with different stream characteristics.

4 Resource-aware Mining Operators

4.1 Operators for Data Streams

The techniques proposed in this work are implemented on top of a Data Stream Management System (DSMS) called PIPES [Krämer and Seeger, 2004]. Rather than a monolithic DSMS, PIPES is an infrastructure that, in conjunction with the comprehending Java library XXL [d. Bercken *et al.*, 2001], allows for building a DSMS specific to concrete applications with full functionality. Usually, DSMS manage multiple continuous queries specified by operator graphs, which allows for reusing shared subqueries. PIPES adapts this concept and introduces three types of graph nodes: sources, sinks and operators, where operators combine the functionality of a source and a sink with query processing functionalities. The resulting query graphs can be build and manipulated dynamically using an inherent publish and subscribe mechanism. This offers, among others, the possibility to adaptively optimize the processing according to resource awareness. PIPES provides the operational run-time environment to process and optimize queries as well as a programming interface to implement new operators, sources and sinks.

The aimed resource awareness mainly arises from implementing the mining techniques as separate PIPES operators. But why do we implement them as operators, rather than, for instance, building specialized DSMS for clustering and frequent itemset mining? The answer is, we want to be able to freely choose among any combination of these mining techniques between themselves, with other mining algorithms and, generally, with all operators implemented in PIPES. Fig. 2 pictures a small example to illustrate this approach (each operator is pictured by its corresponding algebra symbol). The stream data produced by two sources O_1 and O_2 is processed in three ways: sink S_1 receives all clusters determined (ζ) for O_1 after a preprocessing filter step. S_2 and S_3 receive association rules determined (Φ) on joined data from O_1 and O_2 – this implies finding the frequent itemsets (ϕ). S_3 works on the determined rules directly, while S_2 receives them after applying another clustering step (ζ) in order to identify interesting rules by grouping the related ones (similar to the approach in [Toivonen *et al.*, 1995]).

The frequent itemset mining operator takes three dynamically adjustable parameters:

1. The size b of a batch.
2. The queried time interval $[t_s, t_e]$.
3. An output interval o.

b represents the finest granularity of observed time and is equal to the internal update interval u. Thus, o should be a multiple of b. The resulting output is a data stream consisting of one stream element per passed output interval, containing the frequent itemsets found in $[t_s, t_e]$. In correspondence to the aimed resource awareness a user can decide between two possibilities to initialize the operator by providing:

1. Claimed qualities.
2. A memory limit.

In the first case, the amount of memory is calculated by adapting parameters inside the operator to achieve the claimed quality. For the second case, the adapting technique tries to find the ideal parameter settings to achieve optimal quality results, if possible, while adhering to the given memory limit. In order to achieve this, the user must define which of the supported qualities is prioritized and/or weight them accordingly.

4.2 Frequent Itemset Mining

We record two main requirements of frequent itemset mining techniques in order to lend themselves for our needs: they are able to provide error guarantees (frequent itemset mining on data streams usually produces approximate results, e.g., there may be some false positives in the resulting output), and the approach has to be time-sensitive. The FP-Stream approach in [Giannella *et al.*, 2003a] is capable to satisfy these requirements. Asked for the frequent itemsets of a time period $[t_s, t_e]$, FP-Stream guarantees that it delivers all frequent itemsets in $[t_{s'}, t_{e'}]$ with frequency $\geq \sigma \cdot W$, where W is the number of transactions the time period $[t_{s'}, t_{e'}]$ contains. $t_{s'}$ and $t_{e'}$ are the time stamps of the used tilted time window table (TTWT) that correspond to t_s and t_e, depending on Q_{Tg}. The result may also contain some itemsets whose frequency is between $(\sigma - \varepsilon) \cdot W$ and $\sigma \cdot W$.

Our first goal was to find out how much memory the algorithm needs in order to deliver results in a certain quality. We conducted an extensive series of tests for the algorithm's memory requirements in different parameter settings. Secondly, we extended the approach from [Giannella *et al.*, 2003a] so we can cope with limited memory, resulting in the algorithm's resource and quality awareness.

Estimating the amount of memory

Firstly, we estimate the amount of memory a pattern tree needs within a given parameter setting. For estimating the overall memory requirements of a pattern tree, we need to know the number of nodes in a pattern tree, and the amount of memory each individual node needs. In [Giannella *et al.*, 2003a] an upper bound is given for the size of a TTWT by $2 \lceil log_2(N) \rceil + 2$, where N is the number of batches seen so far. This is because the algorithm uses a logarithmic TTWT to the basis 2 and is designed with one buffer value between each two values except the two most recent. In our experiments the actual number of entries averaged over all TTWTs in a tree was always less than 70% of this upper bound. Besides the size of the TTWT, each node in a pattern tree needs some fixed sized memory c for storing the itemset it represents and information about its parent node and child nodes.

The number of nodes in a pattern tree depends on several facts. On the one hand there are the values of the algorithm's input parameters ε and b. The value of σ does not affect the number of nodes in a pattern tree, because the adding and dropping conditions in a pattern tree depend only on ε, though σ is a quality measure! On the other hand there are the characteristics of the underlying data stream like the average number of items per transaction and the number of distinct items in the stream. We have not yet found a concrete formula describing the maximum number of nodes in a pattern tree for specific parameter settings. But, we conducted a series of tests showing that the maximum number of nodes remains constant over time while

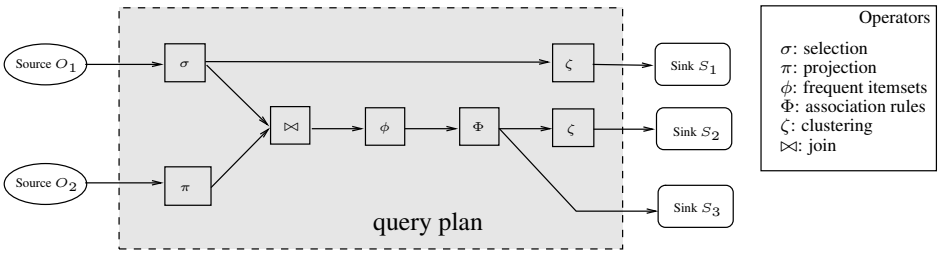


Figure 2: Example query plan

the algorithm processes an infinite series of transactions. Thus, for certain values of ε and b and specific characteristics of the underlying data stream we know the maximum number of nodes in a pattern tree. In general, we can state that a large pattern tree leads to mining results with better quality than a small pattern tree.

The fact that the overall space requirements stabilize or grow very slowly as the stream progresses was already shown in [Giannella *et al.*, 2003a]. The authors investigated different values for σ and the average number of items per transaction. [Giannella *et al.*, 2003b] additionally showed the same effect for varying values of ε. We extended these tests to different values of the number of distinct items in the stream and the size of b. With a constant input rate the size of b affects the number of transactions a batch contains. As we will show in Sect. 5 the value of b is the only one that has very little impact, as long as the number of transactions in a time interval of size b exceeds a certain threshold (depending on ε and some data stream characteristics). This is in order to fade out the effect of temporarily frequent itemsets that have no significance for the overall mining result.

According to Sect. 3 we can determine r as: *number-of-nodes*$(\varepsilon,\, b) \,*\, [2 * \lceil log_2\,(N)\rceil + 2 + c]$, representing a heuristic approximation of the resources actually needed.

Dynamic tree size adjustment

In the following we introduce an extended approach of [Giannella *et al.*, 2003a] that can cope with limited memory and uses the available memory as effective as possible. If there is not enough memory available for finding frequent itemsets in the claimed quality, we need to dynamically adjust the size of the pattern tree according to the actual memory conditions. At first we implemented the approach in [Giannella *et al.*, 2003a] and examined its memory requirements for different parameter settings. After that, we considered a couple of possibilities to control the memory requirements of the pattern tree. Thus, we gained an extended approach that has several alternatives for controlling the tree size depending on the user's quality weighting. We introduce a memory filling factor f defined as follows: $f = \frac{\textit{actual memory usage}}{\textit{provided memory}}$. Depending on f, our approach takes action to reduce or increase the size of the pattern tree. The possibilities for manipulating the size of the tree are:

1. Adjust the value of ε while keeping σ constant, i.e., change the approximation quality Q_{Ma} of our mining results according to available memory.
2. Adjust the value of σ according to available memory while keeping ε/σ constant, i.e., keep the quality Q_{Ma} of the mining result constant but change the quality of interestingness Q_{Mi}.
3. Limit the size of each TTWT according to available memory, i.e., change the time range quality Q_{Tr}.
4. Adjust the value of b according to available memory, i.e., change the time granularity quality Q_{Tg}.

Since we can estimate the memory requirements of a pattern tree for a given set of parameters, we can also estimate the maximum number of nodes our pattern tree may not exceed in order to adhere to a certain memory limit. In each step we assume the size of a TTWT to be at its upper bound. The changes in this upper bound are estimated only at constant intervals (in our tests every 100 batches), because we want to avoid registering negligible changes of the maximum number of nodes after every batch.

Option 1: Adapting ε. A first option for manipulating the size of the pattern tree is to change the value of ε, i.e., change Q_{Ma}. Therefore, we estimate an ideal ε that results in a pattern tree with approximate as many nodes as possible, rather than taking the value of ε as an input parameter. However, the user may specify a lower bound for the value of ε, i.e., an upper bound for the quality of the mining result.

Since we are not yet able to calculate the maximum number of nodes for a given amount of memory exactly, we also cannot estimate an ideal value of ε. Our algorithm adjusts the value of ε depending on the filling factor f after every processed batch as follows:

- $f < 0.85$: Decrease ε by ten percent. Use this ε when processing the following batches.
- $0.85 \leq f \leq 1.0$: The value of ε remains fixed.
- $f > 1.0$: Increase ε by ten percent. Conduct tail pruning at the TTWTs of each node in the pattern tree and drop all nodes with empty TTWTs. Repeat these steps as long as $f > 1.0$.

Note that f can become greater than 1. This is because we assume that we have more memory available in the system than the amount we provide to the pattern tree. $f > 1$ indicates that the pattern tree consumes more memory than we granted to it and the tree will thus reduce its size.

In this approach we have to store the value of ε we used in each specific time interval, in addition to monitoring the number of transactions in each interval. If two TTWT frequency entries n_i and n_j are merged, we also have to merge the according ε values ε_i and ε_j in order to determine the achievable quality for this time period. We can average the two distinct values of ε as described by equation 1. w_i and w_j denote the sizes of time intervals t_i and t_j.

$$\hat{\varepsilon} = (\varepsilon_i w_i + \varepsilon_j w_j) \,/\, (w_i + w_j) \qquad (1)$$

It was shown in [Giannella *et al.*, 2003a] that for a fixed ε, if all itemsets whose approximate frequency is larger than $(\sigma - \varepsilon) \cdot W$ are requested, then the result will contain all frequent itemsets in the period $[t_{s'}, t_{e'}]$. Thus, in our extended approach all itemsets with approximate frequency $(\sigma - \hat{\varepsilon}) \cdot W$ are returned. The value of $\hat{\varepsilon}$ depends on the time intervals contained in $[t_{s'}, t_{e'}]$. If $[t_{s'}, t_{e'}]$ covers only time intervals where the same value of ε was used for all

batches, then $\hat{\varepsilon}$ is equal to this ε. If $[t_{s'}, t_{e'}]$ contains time intervals where the value of ε differs, the value of $\hat{\varepsilon}$ becomes:

$$\hat{\varepsilon} = \left(\sum_{i=s'}^{e'} \varepsilon_i w_i \right) / W \quad (2)$$

In summary, in our first option we control the amount of memory used by varying the value of ε and so ε/σ, which reflects the approximational quality Q_{Ma} of our mining result.

Option 2: Adapting σ. The second option for adjusting the size of the pattern tree is to alter the value of σ. As σ does not influence the size of the tree directly, ε/σ remains the same. That is why the user does not provide a fixed value of σ, but rather claims for a certain ε/σ that should be guaranteed. Again, the user may also specify a lower bound for the value of σ. The handling of σ is analog to the handling of ε in the first option with the only difference, that ε must be adjusted in parallel in order to keep ε/σ constant. The analogy also applies for determining the value of $\hat{\sigma}$:

$$\hat{\sigma} = \left(\sum_{i=s'}^{e'} \sigma_i w_i \right) / W \quad (3)$$

The value of $\hat{\varepsilon}$ again results from equation 2. Returned are all itemsets whose approximate frequency is larger than $(\hat{\sigma} - \hat{\varepsilon}) \cdot W$. We guarantee to deliver all itemsets whose actual frequency is $\geq \hat{\sigma}$. Because $\hat{\varepsilon}/\hat{\sigma}$ is kept constant as requested by the user, the quality Q_{Ma} meets the user's requirements. By adjusting the value of σ we alter the quality Q_{Mi} by varying the frequency an itemset must occur in order to belong to the delivered set of results.

But, what is the difference between the first and the second option? In the first option we keep σ constant and change ε. Thus, we can request all itemsets with a minimum support of $(\sigma - \hat{\varepsilon}) \cdot W$ and accept a poorer quality Q_{Ma}. In the second option, we modify σ but keep ε/σ constant. Thus, we also keep the user defined quality Q_{Ma} constant and return all itemsets with a minimum support of $(\hat{\sigma} - \hat{\varepsilon}) \cdot W$. In this case, only the more interesting itemsets are found, in terms of σ as an interestingness measure from Q_{Mi}. An effect on memory usage is achieved in both options. Moreover, in both options the methodology of pruning TTWTs remains unchanged.

Option 3: Limiting the size of the TTWTs. The third option is to limit the size of each TTWT to a fixed value. This results in a restriction of how far we can look back into the history of the processed stream, because we limit the number of time intervals for which we store frequency information. Thus, we are not able to deliver results from a time period that includes batches lying farther back in history than the information we recorded. According to Sect. 3 this leads to an impairment of the time range quality Q_{Tr}.

As the number of time intervals stored in one entry of a TTWT increases logarithmically, saving a large amount of memory demands for limiting the size of a TTWT drastically. In this way, the information of a considerable portion of the observed batches would get lost. Lowering the maximal size by small values, only a small amount of memory can be saved.

Option 4: Adapting b. The last option in order to limit the used memory is to adjust the value of b. Assuming a constant input rate, the size of b affects the number of transactions a batch contains. Increasing b leads to an impairment of the time granularity quality Q_{Tg}, as we reduce how exact we can look back. Our experiments reveal that the number of transactions per batch does not affect the number of nodes in the pattern tree significantly. By increasing the value of b we can only reduce the total number of entries in a TTWT while processing a finite part of the stream. Considering infinite data streams, since the size of a TTWT grows logarithmically, the number of entries in a TTWT will converge to the same value for all choices of b. Therefore, this option will only have a significant impact on the required memory if it is combined with a limitation of the TTWT size. Combining both, we can reduce the granularity of a time interval and look back farther in the history of the data stream using the same number of TTWT entries.

4.3 Change Detection

In Sect. 1, we briefly discussed the importance of quickly detecting changes in data streams. In the following we will exemplarily discuss change detection on the basis of the introduced frequent itemset mining algorithm. We will outline how change detection may be implemented efficiently and, more important, resource-aware. In the next section we will investigate how our approach can be mapped to general stream mining tasks, including change detection as a post-processing step on the basis of concrete mining tasks as well as an independent mining problem. On the basis of the extended FP-Stream approach there arise several possibilities in order to implement an efficient detection of changes in the frequent itemsets themselves, their temporal occurrences and/or other relevant aspects. With several of these alternatives we even get a resource-adaptive approach for detecting changes for free. We will sketch five different approaches and briefly discuss pros and cons.

From our point of view any technique for change detection should meet the following criteria:

- it is based on data structures which can be limited in size, preferably in a dynamic manner
- it produces realtime results, which requires an online and incremental processing
- it is capable of detecting any kind of changes that may be of interest (i.e., frequency changes, correlation changes, temporal changes, and so on) and is not restricted to the detection of specific kinds
- changes can be represented to the user in an intuitive and understandable manner

With the problem of frequent itemset mining, a lot of *sophisticated* changes in the stream (e.g., if subsets of a formerly frequent itemset are still frequent) can be detected by analyzing *basic* changes (i.e., the occurrence/omission of single members of the set of all frequent itemsets). Therefore, for now we primarily focus on detecting these basic changes.

Approaches on Separate Data Structures

A naive but simple solution is to take the output of the frequent itemset mining operator as an independent input stream for a separate change detection operator. Like this, information about the itemsets found in the past must be stored in some separate data structure. We investigated three different approaches for that:

1. store all frequent itemsets produced so far together with additional information (e.g., temporal) in a table

2. store snapshots of the pattern tree in regular intervals
3. store only differences between the set of all frequent itemsets in regular intervals (similar to the incremental backup technique for database systems)

A main problem of all three approaches is the need for a separate data structure. This data structure allocates extra resources, and thus, resource limits must be shared between the actual mining operator and the change detection operator. However, the resource-adaptive techniques introduced in this work could be applied to this data structure in order to meet given resource limits. The first two methods allow for detecting a wide variety of changes, even temporal ones, but are consuming by far too much memory. With the first variant, the task of detecting changes in one specific itemset is complicated, because there is no information about the location of itemsets in the pattern tree if they are registered after being output from the frequent itemset operator. The last of the three methods is suitable for quickly detecting new or omitted itemsets between two successive time steps – but complicates the handling of arbitrary time intervals and the detection of changes in the frequency of itemsets that have already been frequent before.

Approaches on the Pattern Tree

In order to implement a resource-efficient change detection, a more intuitive approach is to try to detect changes directly from the pattern tree used in the frequent itemset mining operator. This would need no extra memory, which is one of the main resources in our considerations. Moreover, because change detection and itemset mining is combined into one operator, realtime signalizing is easy to achieve. Again, we distinguish two specific approaches:

1. detect changes as soon as the pattern tree is modified
2. detect changes after executing the FP-growth algorithm on the pattern tree (when looking for the actual frequent itemsets after each batch)

The second approach is capable of detecting more changes, because with the first one we can only watch itemsets that are modified in the current batch run. If a change in an itemset is only recognized during the run of FP-growth, the first method will not detect this change. The disadvantage of the second method is that it can only be run after processing a whole batch and has to completely traverse the tree – which leads to worse reaction time and less information about new or deleted nodes. However, in the first method we have to deal with *change candidates*, because not each modification will lead to a actual change in the itemsets. Advantages of both approaches, in contrast to those on separate data structures, are low runtime, easy detection of temporal changes (the TTWTs containing temporal information can be analyzed directly) and no extra memory consumption.

Usually, the reaction time Q_{T_c} is the most important quality measure for change detection in data streams. The approaches working on the pattern tree, particularly the first one, can provide better values for Q_{T_c} than those on separate data structures, because it does not depend on the output interval of the mining operator or a predefined interval. Rather, both techniques can be integrated right into the frequent itemset mining operator.

In this section, we briefly presented preliminary results we achieved when investigating approaches for resource-aware change detection. For now, we state that the choice of algorithm depends on the goals actually desired by the user. Under special circumstances several of the introduced methods should be combined – when aiming for a resource-aware, and moreover, resource-efficient, method, those based on the pattern tree should be preferred. In future works we will investigate approaches for change detection more detailed, including the combination with resource-adaptive techniques, considerations about achievable qualities (mainly for Q_{T_c}, but also other measures like Q_{T_g} and Q_{M_a} have to be considered) as well as change detection methods for general mining tasks.

4.4 General Applicability of the Approach

We are currently generalizing our approach of resource-adaptive frequent itemset mining. Our work aims at making the technique applicable to data stream mining algorithms with certain characteristics in general.

To start with, there exist other algorithms for mining frequent itemsets in data streams that use the same approach of approximated frequencies as the FP-Stream algorithm, e.g., [Manku and Motwani, 2002; Chang and Lee, 2004]. We can thus extend these algorithms in a way analog to the modification of FP-Stream. In general, each algorithm whose resource requirements depend on parameters like σ and ϵ can be modified like that – with different impact on the grade of adaptiveness.

Some of the presented methods to manipulate the size of the pattern tree can be applied to these algorithms without major changes to both method and algorithm. Despite the usage of different data structures in these algorithms, it is possible to adapt the size of their synopsis by varying the values of σ and ϵ. The quality of the mining results will remain a computable value that can be guaranteed like in the extended FP-Stream algorithm.

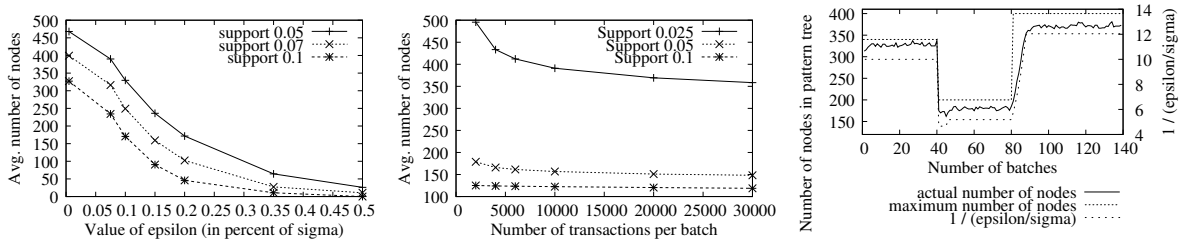
A majority of data stream mining algorithms uses synopsis data structures to capture the content of the data stream. Since these synopses only represent an approximation of the stream's actual content, there is always the notion of quality associated with it. Our method can thus be applied in principle to such stream mining algorithms as well. In [Franke *et al.*, 2005] we already successfully applied the approach in order to implement a resource-aware clustering operator for PIPES.

5 Evaluation

The purpose of the following evaluation is twofold: at first we will show how we achieve the aimed quality awareness. To show this, we will evaluate how good the algorithms achieve a claimed quality and how exact the corresponding calculations are. In the context of the proposed stream mining approach quality awareness comes along with the adaption to provided resources and the determination of needed resources concerning aimed quality measures. This is the second objective of this section: we will show that memory requirements are approximated satisfyingly and that they are met finally, and which conclusions we can draw to the achieved quality.

Before evaluating our quality-aware frequent itemset mining approach we conducted a series of tests with the original FP-Stream algorithm. We figured out how the algorithm behaves for different parameter settings of ε, σ and b. We used synthetic data generated by the IBM market-basket data generator. In the first set of experiments 1M transactions were generated using 1K distinct items and an average transaction length of 3. All other parameters were set to their default values from the generator.

Since in the original approach a batch does not cover a constant period of time but a constant number of transac-



(a) Varying support and epsilon (b) Varying support and batch size (c) Resource awareness

Figure 3: Average number of nodes in a pattern tree

tions, we set the size of a batch to 5000 transactions. Figure 3(a) shows some of our experimental results with the original algorithm. We measured the average number of nodes in a pattern tree for three different values of σ. For each σ we run the algorithm with various ratios between ε and σ to simulate different quality demands. As expected the number of nodes decreases with rising ε and σ.

Figure 3(b) shows results of another series of tests we conducted with the original algorithm. We processed the test data with different values of σ (always setting $\varepsilon = 0.1 \cdot \sigma$) and with various batch sizes. The number of nodes in the pattern tree does not decrease significantly when the batch size is highly raised.

We also processed test data having more average items per transactions (5, 10) and/or more distinct items (5K, 10K) and we additionally tried different batch sizes (1000 to 30000). The main conclusion is always as stated above. One remarkable thing we noticed is that for small batch sizes the number of nodes in a pattern tree is far from being constant. For example, when processing testing data with 1M transactions, 1K distinct items and average transaction length of 3 we set $\sigma = 0.025$, $\varepsilon = 0.1 \cdot \sigma$ and the batch size to 1000 transactions. The number of nodes in the resulting pattern tree oscillated between under 1000 to nearly 5000. When processing the same data set with higher values of σ, ε or batch size this range got significantly smaller. For a support of 0.07 the difference between the minimum and the maximum value of the number of nodes was 45. Remainders of this effect can be seen in Figure 3(b) for $\sigma = 0.025$ by the sharp decrease of the number of nodes for low batch sizes.

For evaluating the quality-aware frequent itemset mining approach, we again generated data sets with 1M transactions using 1K distinct items and an average transaction length of 3. Our algorithm consumed the stream of transactions from a source with a constant output rate of 3 seconds. The value b of the finest granularity of time was set to 15000 seconds, so each processed batch contained 5000 transactions. The value of σ was set to 0.05.

Firstly, we wanted to demonstrate that our approach can cope with changing memory conditions. Instead of limiting the actual amount of available memory we limited the number of nodes that the pattern tree may have. One could calculate the actual amount of memory needed, since the maximum size of a TTWT can be estimated as described in Section 4.2.

Initially we limited the maximum number of nodes the generated pattern tree may have to 340. As we do not have a formula yielding the maximum number of nodes in a pattern tree for a given set of parameters, we had to choose an adequate value of ε for the algorithm. Our previous experiments showed, that $\varepsilon = 0.1 \cdot \sigma = 0.005$ would be a good value to start.

We started the algorithm and decreased the maximum number of nodes after 40 batches to 200 nodes. We then raised it again after 40 batches to a value of 400 nodes. Figure 3(c) shows the algorithm's behavior. After 40 batches, it had to raise the value of ε several times to get its filling factor below 1. Then, after the next 40 batches, the algorithm lowered the value of ε after every processed batch until a tree size was reached that was close enough to the maximum according to the filling factor. Figure 3(c) also displays the quality of the stream mining for every batch, i.e., the value of $1/(\varepsilon/\sigma)$.

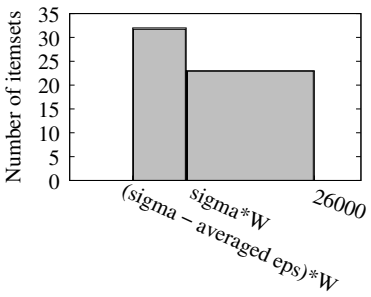


Figure 4: Quality awareness

With our second series of tests of our quality-aware approach we demonstrate the quality of our mining results. We started the algorithm using option 1 of our proposed techniques for adjusting the size of the pattern tree, i.e., we change the value of ε when necessary. The experimental settings are equal to these of the above test. Only this time we set $\sigma = 0.025$ and $\varepsilon = 0.2 \cdot \sigma$ to get any frequent itemsets at all. The experiments show that we estimated the overall quality of our mining results ranging over several batches (and thus several values of ε) correctly and that it is in fact suitable to average distinct values of ε as described.

First we run our approach once to get the value of $\hat{\varepsilon}$, which was approximately 0.00765. Then we used the original FP tree algorithm to get the actual set of frequent itemsets from our test data. We also got the actual frequencies of all itemsets having frequencies between $(\sigma - \hat{\varepsilon})$ and a little less and the required support. Then, we ran our quality-aware approach and asked for the set of frequent itemsets after the 120th batch. We set the time period $[t_s, t_e]$ to the whole period of time the stream was processed so far. So the window we requested contained $W = 600000$ transactions. After we received the result, we compared the output to what we knew from the FP-growth method. For each itemset our approach delivered, we looked up its real frequency and printed these information in the histogram shown by Figure 4. All itemsets our approach output were inserted in frequency buckets, according to their real frequency which we gained through using the FP tree algorithm.

Figure 4 shows that the output contained no itemsets having frequency less than $(\sigma - \hat{\varepsilon}) \cdot W$. The histogram also shows that exactly 23 itemsets having frequency of at least σ were delivered. These itemsets are the same as FP-growth delivered.

All in all the experiments met our expectations. In [Franke *et al.*, 2005] we also conducted similar experi-

ments for a resource-aware clustering strategy, which emphasize the results showed here. The approximations of memory usage hold, the quality deduced from the available resources is close to the actually achieved quality. This is true for all examined quality measures, as far as our tests can show that. Of course, we have to do a couple of extended test series, including different parameter settings, varying stream characteristics and a deeper analysis of several (classes of) quality measures. With the results of these subsequent tests we could finally demonstrate the quality and resource awareness already achieved in this work.

6 Conclusion

In this paper, we argued that quality of analysis results is an important issue of mining in data streams. The reason is that stream mining can be usually performed only on a resource-limited subset or approximation of the entire stream, which affects different measures of data quality. Based on specific quality measures for stream mining, we investigated and enhanced a frequent itemset mining technique in order to estimate the quality depending on the current resource situation (mainly the available memory) as well as to allocate resources needed for guaranteeing user-specified quality requirements. Furthermore, we gave directions for making a whole class of stream mining algorithms resource- and quality-aware, including complementary tasks such as application specific change detection.

Beside these goals and the mentioned considerations about resource-aware approaches for change detection in data streams we will also apply the introduced techniques to other mining primitives. Based on the earned experiences about the method's practical applicability and the reflections on different quality measures we plan to build a formal framework for resource-adaptive stream mining under quality guarantees. This includes the definition of the concrete quality/resource functions r and $\overline{r}$.

From our point of view, other resource requirements than memory consumption of stream synopses have to be regarded as well. For instance, beside the memory required by the pattern tree itself, the algorithm needs to have additional memory available during runtime. This memory is used for the actual computations and for storing two batches, the one that is currently processed as well as the one that is currently build from the newly arriving transactions. When considering the computations done by the FP-Stream algorithm, we note that the FP-growth method used to determine the batch's frequent itemsets is very expensive in terms of memory requirements. Using a more memory efficient FP-growth method as proposed and implemented by Özkural and Aykanat [Özkural and Aykanat, 2004] would lead to decreased overall memory requirements of the algorithm.

Moreover, in addition to memory awareness we will take the algorithms' runtime into account. In this context several additional aspects have to be considered, like the streaming rate of the incoming data and the runtime overhead imposed by our extension. In addition, we will have to deal with the conflict that adding resource adaptiveness to an algorithm imposes a runtime and memory overhead itself.

Finally, we have addressed only operator-locally parameters like the amount of memory available for this specific operator. In future work, we plan to take also global properties of the whole analysis pipeline into account, e.g. load shedding and windowing operators, which have an impact on the output quality, too.

References

[Berti-Équille, 2006] L. Berti-Équille. Contributions to Quality-Aware Online Query Processing. *IEEE Data Eng. Bull.*, 29(2):32–42, 2006.

[Chang and Lee, 2004] J. H. Chang and W. S. Lee. A Sliding Window Method for Finding Recently Frequent Itemsets over Online Data Streams. *Journal of Information Science and Engineering*, 20(4):753–762, 2004.

[d. Bercken *et al.*, 2001] J. V. d. Bercken, B. Blohsfeld, J.-P. Dittrich, J. Krämer, T. Schäfer, M. Schneider, and B. Seeger. XXL - A Library Approach to Supporting Efficient Implementations of Advanced Database Queries. In *VLDB 2001*, pages 39–48, 2001.

[Franke *et al.*, 2005] C. Franke, M. Hartung, M. Karnstedt, and K. Sattler. Quality-Aware Mining of Data Streams. In *IQ*, 2005.

[Gaber and Yu, 2006] M. M. Gaber and Ph. S. Yu. A framework for resource-aware knowledge discovery in data streams: a holistic approach with its application to clustering. In *SAC'06*, pages 649–656, 2006.

[Gaber *et al.*, 2003] M. M. Gaber, Sh. Krishnaswamy, and A. Zaslavsky. Adaptive mining techniques for data streams using algorithm output granularity. In *The Australasian Data Mining Workshop*, 2003.

[Giannella *et al.*, 2003a] C. Giannella, J. Han, J. Pei, X. Yan, and P. S. Yu. Mining Frequent Patterns in Data Streams at Multiple Time Granularities. In *Workshop on Next Generation Data Mining*, 2003.

[Giannella *et al.*, 2003b] C. Giannella, J. Han, E. Robertson, and C. Liu. Mining Frequent Itemsets over Arbitrary Time Intervals in Data Streams. Technical report, Indiana University, 2003.

[Han *et al.*, 2000] J. Han, J. Pei, and Y. Yin. Mining Frequent Patterns without Candidate Generation. In *SIGMOD 2000, Dallas, USA*, pages 1–12, 2000.

[Krämer and Seeger, 2004] J. Krämer and B. Seeger. PIPES - A Public Infrastructure for Processing and Exploring Streams. In *SIGMOD 2004*, pages 925–926, 2004.

[Manku and Motwani, 2002] G. S. Manku and R. Motwani. Approximate Frequency Counts over Data Streams. In *VLDB 2002, Hong Kong, China*, pages 346–357, 2002.

[Özkural and Aykanat, 2004] E. Özkural and C. Aykanat. A Space Optimization for FP-Growth. In *ICDM Workshop on Frequent Itemset Mining Implementations*, 2004.

[Tan and Kumar, 2000] P. Tan and V. Kumar. Interestingness Measures for Association Patterns: A Perspective. Technical report, University of Minnesota, 2000.

[Teng *et al.*, 2004] W.-G. Teng, M.-S. Chen, and Ph. S. Yu. Resource-aware mining with variable granularities in data streams. In *SDM 2004*, 2004.

[Toivonen *et al.*, 1995] H. Toivonen, M. Klemettinen, P. Ronkainen, K. Haton, and H. Mannila. Pruning and grouping discovered association rules. In *ECML Workshop on Statistics, Machine Learning and Knowledge Discovery in Databases*, pages 47–52, 1995.